Parallel Programming Laboratory 4

Josue Peña A., Jeffrey García G.

May 2020

Contents

1 Introduction **1**

2 Problem description **1**

2.1 MPI solution 2

2.2 CUDA solution 2

2.2.1 Strategy 2

3 Machine specifications **2**

4 Analysis & conclusions **3**

1 Introduction

In this lab assignment we choose to parallelize the matrix multiplication algorithm, which was implemented in an earlier class laboratory using the parallel library MPI in the C programming language. The reason for this was so that we could exemplify the performance and code simplicity gains that come with using pyCUDA.

2 Problem description

The algorithm we choose for this test was matrix multiplication. We decided to implement it since it is an integral of something we had worked with previously in our Scientific Computing class, the Cholesky Decomposition method, which in turn is an integral part of the least squares problem. The least squares problem approximates the solution for undetermined systems of equations (sets of equations in which there are more equations than unknowns). We then plan to implement the least squares problem for the course's protect.

It's important to note that in this particular setting, all the matrices used in Cholesky decomposition are always square, so we only take this case into account in our algorithm.

2.1 MPI solution

We used blocking and non-blocking communication functions for our design (functions *MPI_Isend()*, *MPI_Irecv()*, *MPI_Wait()*, etc). Inside the for loop which iterates over all rows of the C matrix, right after the lines of code for processing a single row, the root processor waits (Using *MPI_Wait()*) until two buffer variables are ready for receiving. These two variables are, respectively, the index of the processed row and the array with it's contents.

2.2 CUDA solution

For the application of the CUDA solution it is important to know the base concepts of CUDA programming and how it performs the parallelization using the kernel. For this approach we use PyCUDA that is wrap of the CUDA API to perform parallel computation in the Python 2 programming language.

2.2.1 Strategy

Firstly, we did the basic steps for PyCUDA programming which is to receive the matrix dimensions input from the user, generate the corresponding random matrices and pass them to the GPU memory through the command *gpuarray.to_gpu*.

Then we focus on the execution of the main activities that occur within the kernel. We set a 2D block of 32x32 threads that will be running within a singular 2D dynamic size block grid which will vary depending on the input matrix dimensions. Also it is important to note that the matrix is represented as an array in the gpu so the indexation changes accordingly and it is shown so in the code.

To access the matrix from the kernel we calculate the indexes with these operations $tx * \%(MATRIX_SIZE)s + k$ and $k * \%(MATRIX_SIZE)s + ty$ representing the matrix indexes in row order and matrix indexes in column order, being tx and ty the thread iterators for each element on the matrix. So performing with these operations the matrix multiplication for each cell that stores its result in the correspondent index of the output matrix passed as an argument to the function kernel.

3 Machine specifications

- CPU: Intel i3-3220
- CPU cores: 2
- GPU: Nvidia gt 740
- GPU standard memory: 1024 MB GDDR5

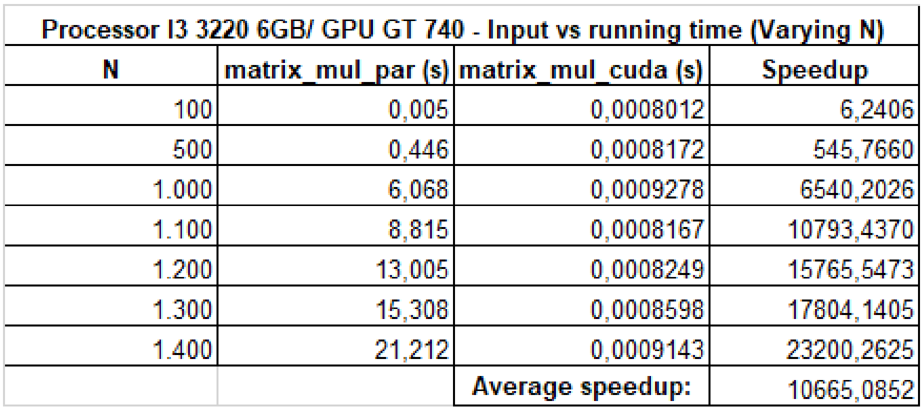
- CUDA cores: 384
- GPU bandwidth: 80.19 GB/s
- RAM: 6 GB

4 Analysis & conclusions

At first the results on the speedups for this assignment might seem to be too exaggerated. It's important to point out the significant difference in computing power and optimization from between the two parallel approaches.

In our MPI solution, we only had access to 2 CPU cores and the communication and synchronization schemes were designed by us (so they're not the most efficient or optimal ways of solving this problem).

By contrast, PyCuda handled all of these constraints by itself in the most efficient way possible, and most importantly, it had 384 shading units or CUDA cores to do so. If the we could had harnessed the an equivalent amount of computing power for the MPI solution then the speedups from the pyCuda approach would had been more fair.



Processor I3 3220 6GB/ GPU GT 740 - Input vs running time (Varying N)			
N	matrix_mul_par (s)	matrix_mul_cuda (s)	Speedup
100	0,005	0,0008012	6,2406
500	0,446	0,0008172	545,7660
1.000	6,068	0,0009278	6540,2026
1.100	8,815	0,0008167	10793,4370
1.200	13,005	0,0008249	15765,5473
1.300	15,308	0,0008598	17804,1405
1.400	21,212	0,0009143	23200,2625
		Average speedup:	10665,0852

Figure 1: Table of the different running times for each program

In Figure 1 we can see the running times for the MPI and pyCUDA algorithms with varying sizes of square NxN matrices. As the matrices' sizes were larger, the speedup increase grew rapidly larger as well.

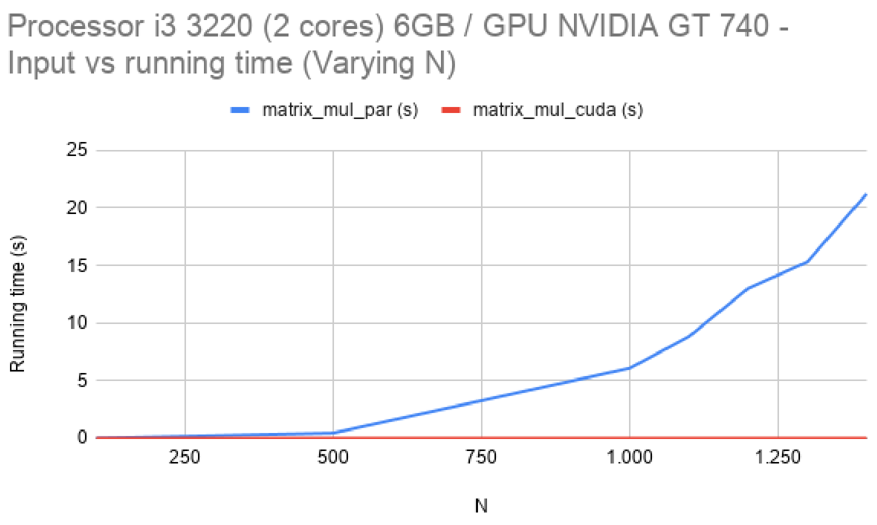


Figure 2: Plot of the matrix size vs the running time for each program

A design limitation for the implemented PyCuda kernel function is that the matrices' dimensions have to be smaller than 1400x1400. If not, memory allocation behaves unexpectedly and there might be illegal memory accesses. This probably may be caused by the thread-block-grid configurations which are constrained by the hardware characteristics and limitations of our machine's GPU and many low-level CUDA concepts that we don't fully dominate yet as of today.

As conclusion, PyCUDA is an excellent and powerful tool for performing parallel computing on a GPU. But it's level of complexity and relative lack of high quality online information makes CUDA programming a very tough programming task.